Optimizing the Induction of Alternating Decision Trees

Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby

University of Waikato, Hamilton, New Zealand,
{bernhard, geoff, rbk1}@cs.waikato.ac.nz,
WWW home page: http://www.cs.waikato.ac.nz/~ml

Abstract. The alternating decision tree brings comprehensibility to the performance enhancing capabilities of boosting. A single interpretable tree is induced wherein knowledge is distributed across the nodes and multiple paths are traversed to form predictions. The complexity of the algorithm is quadratic in the number of boosting iterations and this makes it unsuitable for larger knowledge discovery in database tasks. In this paper we explore various heuristic methods for reducing this complexity while maintaining the performance characteristics of the original algorithm. In experiments using standard, artificial and knowledge discovery datasets we show that a range of heuristic methods with log linear complexity are capable of achieving similar performance to the original method. Of these methods, the random walk heuristic is seen to outperform all others as the number of boosting iterations increases. The average case complexity of this method is linear.

1 Introduction

Highly accurate classifiers can be found using the boosting procedure but as [6] discovered for standard decision trees the combination of each classifier produced at each iteration into a single classifier is multiplicative. Freund and Mason [4] introduced a method capable of inducing a single classifier without the exponential growth in tree size by changing the representation of the underlying tree.

Standard decision trees have interior nodes that perform tests on the data and leaf nodes labelled with class values. Classification is achieved by following the unique path from the root to a leaf for a given unknown instance. The alternating decision tree introduces a new node called a predictor node which can be either an interior or a leaf node. The tree has a predictor node at its root and then alternates between test node and further predictor nodes, hence the name. Classification is achieved by summing the contributions from the predictor nodes of all paths that an instance successfully traverses. A positive sum implies membership of one class and a negative sum membership of the other. While the original algorithm was restricted to two class problems it appears that the algorithm can be extended to multiclass problems by using the framework of [7].

Each boosting iteration adds a test (weak hypothesis) and two predictor nodes to the tree. The test chosen to extend the tree is the one that minimizes a function that measures the "impurity" of the test. The tree can be extended from any of its existing predictor nodes which means that for each boosting iteration the minimization function must be computed for each possible test, i.e. the algorithm is quadratic in the number of boosting iterations. This paper explores heuristic methods for restricting the number of predictor nodes that need to be examined for the possible addition of new test nodes. By maintaining the performance levels of the original algorithm and reducing its complexity we aim to demonstrate that it is possible to produce a practical form of the alternating decision tree that can be applied to larger knowledge discovery tasks.

The paper is organized as follows. In the next section we outline our interpretations of the original algorithm (some aspects of the induction of alternating decision trees were not clearly defined in the original paper). Section 3 looks at ways in which the original algorithm can be made more efficient with no loss of performance. While these techniques improve the algorithm they do not have any effect on the overall complexity, and so Section 4 introduces three heuristic search mechanisms for constructing useful paths in the tree without exploring all tests at each predictor node. Each of these methods is log-linear in the worst case. Section 5 outlines an experiment to determine the efficacy of the heuristic methods compared to the original. Accuracy, runtime, and "shallowness" of the tree are measured for a range of standard, artificial and knowledge discovery in database datasets. Shallowness is measured as the number of leaves in the tree. This measure gives a picture of the effect the heuristics have on the overall shape of the trees that they prune. Section 6 provides a discussion of the results and outlines some avenues for further work.

2 Inducing Alternating Decision Trees

Alternating decision trees provide a mechanism for combining the weak hypotheses generated during boosting into a single representation. Keeping faith with the original implementation, we use inequality conditions that compare a single feature with a constant as the weak hypotheses generated during each boosting iteration. In [4] some typographical errors and omissions make the algorithm difficult to implement so we include below a more complete description of our implementation.

At each boosting iteration t the algorithm maintains two sets, a set of preconditions and a set of rules, denoted $\mathcal{P}_t$ and $\mathcal{R}_t$, respectively. A further set $\mathcal{C}$ of weak hypotheses is generated at each boosting iteration.

Initialize Set the weights associated with each training instance to 1. Set the first rule $\mathcal{R}_1$ to have a precondition and condition which are both true. Calculate the prediction value for this rule as $a = \frac{1}{2} \ln \frac{W_+(c)}{W_-(c)}$ where $W_+(c)$, $W_-(c)$ are the total weights of the positive and negative instances that satisfy condition c in the training data. The initial value of c is simply True.

Pre-adjustment Reweight the training instances using the formula

$$w_{i,1} = w_{i,0}e^{-ay_t}$$

(for two class problems, the value of y_t is either +1 or -1).

Do for t = 1, 2, . . ., T

1. Generate the set $\mathcal{C}$ of weak hypotheses using the weights associated with each training instance $w_{i,t}$

2. For each base precondition $c_1 \in \mathcal{P}_t$ and each condition $c_2 \in \mathcal{C}$ calculate

$$Z_t(c_1, c_2) = 2\Big(\sqrt{W_+(c_1 \wedge c_2)W_-(c_1 \wedge c_2)} + \sqrt{W_+(c_1 \wedge \neg c_2)W_-(c_1 \wedge \neg c_2)}\Big) + W(\neg c_1)$$

3. Select c_1, c_2 which minimize $Z(c_1, c_2)$ and set $\mathcal{R}_{t+1}$ to be $\mathcal{R}_t$ with the addition of the rule r_t whose precondition is c_1, condition is c_2 and two prediction values are:

$$a = \frac{1}{2}\ln\frac{W_+(c_1 \wedge c_2) + 1}{W_-(c_1 \wedge c_2) + 1}, \qquad b = \frac{1}{2}\ln\frac{W_+(c_1 \wedge \neg c_2) + 1}{W_-(c_1 \wedge \neg c_2) + 1}$$

4. Set $\mathcal{P}_{t+1}$ to be $\mathcal{P}_t$ with the addition of $c_1 \wedge c_2$ and $c_1 \wedge \neg c_2$.

5. Update the weights of each training example according to the equation

$$w_{i,t+1} = w_{i,t}e^{-r_t(x_i)y_t}$$

Output the classification rule that is the sign of the sum of all the base rules in R_{T+1}:

$$class(x) = sign\left(\sum_{t=1}^{T} r_t(x)\right)$$

The best value of T for stopping the boosting process is still an open research question. In [4] the value is decided by cross-validation. In this paper we look at the effects of heuristics on fixed values for T.

Figure 1 depicts a sample alternating decision tree. A hypothetical example with attribute values $A1 = true$ and $A2 = false$ would be classified according to the following sum derived by going down all appropriate paths in that tree collecting all prediction values encountered: $0.5 + -1.2 + -3.4 + 0.2 = -3.9$ (indicated by horizontal arrows in the figure).

The changes to the original algorithm are fairly minor - the pre-adjustment phase may have been "implicitly" defined, and the change in the Z_t formula to represent all instances that do not satisfy the precondition must be typographical as is the missing minus sign in the updating phase. The formulas for the newly generated predictor nodes in stage 3 have a unit value added to avoid zero-frequency problems [8].

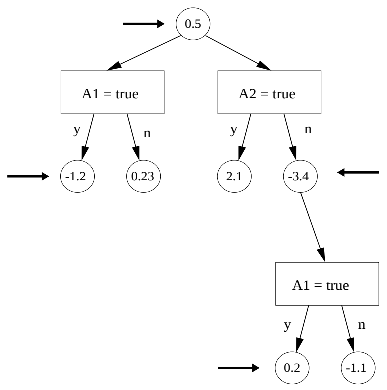

Fig. 1. A sample ADTree. The horizontal arrows indicate all predictor nodes encountered when classifying an example with $A1 = true$ and $A2 = false$.

3 Optimizing the original algorithm

The algorithm described in Section 2 is quadratic in the number of boosting iterations because the calculation of the Z-value for each of the set of $\mathcal{C}$ weak hypotheses is performed at every predictor node in the tree. While this complexity is unavoidable, there are ways to avoid performing the Z-value calculation unnecessarily. We call this value $Z_{pure}(c)$. It is the best possible Z-value that would result from a pure split of the training instances under consideration.

$$Z_{pure}(c) = 2(\sqrt{W_+(c)} + \sqrt{W_-(c)}) + W(\neg c_1)$$

Straightforwardly using the formula for Z given in the previous section would have yielded a lower bound of $Z_{pure} = W(\neg c_1)$ only. But as we adjust all weight sums in a way reminiscent of the Laplace correction, we are able to derive this more stringent lower bound. Z_{pure} is a lower bound on the Z-value a good test could possibly achieve [1]. So if Z_{pure} for some predictor node in the tree is worse

[1] We omit the proof here, but basically one has to show that the following inequality holds for all $a, b, c, d \geq 0$: $\sqrt{a+b+1} + \sqrt{c+d+1} \leq \sqrt{(a+1)(c+1)} + \sqrt{(b+1)(d+1)}$.

than the best test found so far, we do not need to evaluate any test at this node. Furthermore, one can show that all possible tests at all successor nodes of this node are also bounded by the same Z_{pure}, as they involve subsets of the current set of examples only. Therefore we can omit evaluating the complete subtree rooted at a node cutoff by Z_{pure}.

Duplicated tests have been identified as another source of unnecessary inefficiency. Especially with larger numbers of boosting iterations (100 and 200 in our experiments reported below) duplicated tests are reasonably common to justify special attention. If a predictor node is the root of two identical tests, both tests will induce identical subsets when searching for the next best test to add to the tree. Thus we will duplicate work unnecessarily. Fortunately, there is a simple remedy for the problem: when adding a new test to a predictor node, we simply need to check whether exactly the same test is already present at this node. If so, we just merge the old test with the new one by adding the respective prediction values. This procedure results in exactly the same predictive behaviour of the induced alternating decision tree due to its additive nature. But when determining the next best test we save time by traversing a smaller tree.

The effects of both the Z_{pure} cutoff and the *merging of tests* have been studied experimentally and are discussed in Section 5. Summary results are depicted in Figure 3.

4 Heuristic search variants

Even though both methods described in the previous section do improve the efficiency of the algorithm, they do not alter its quadratic nature in general. Determining the next best test to add still involves looking at (almost all) current predictor nodes and for each of those evaluating all possible tests. With every new test, i.e. at every boosting iteration where we are not able to merge tests, we add two more predictor nodes to the tree. A way of reducing the total complexity is to limit the search to just a subset of all predictor nodes, hopefully including the node that would have yielded the next best test using the exhaustive search of the original induction algorithm.

Figure 2 demonstrates the heuristic we have chosen to investigate here. Instead of recursively exploring the complete tree, we limit search at each boosting iteration to just one path down the tree. Obviously this must reduce the complexity, as now we will only be exploring a logarithmically-sized subset of all predictor nodes. Additionally, this procedure seems to yield more shallow trees on average, thus improving efficiency further. On the other hand, such heuristically induced trees are different from the original trees, so we will have to explore whether we are trading off gains in efficiency for worse predictive error or less comprehensible trees, or even both.

The next section will empirically explore these questions, but first we need to define the heuristics used for determining the particular paths to be explored. Basically, we would like to have a good chance of including the node with the best test. In order to achieve this we invented the first two of the following three

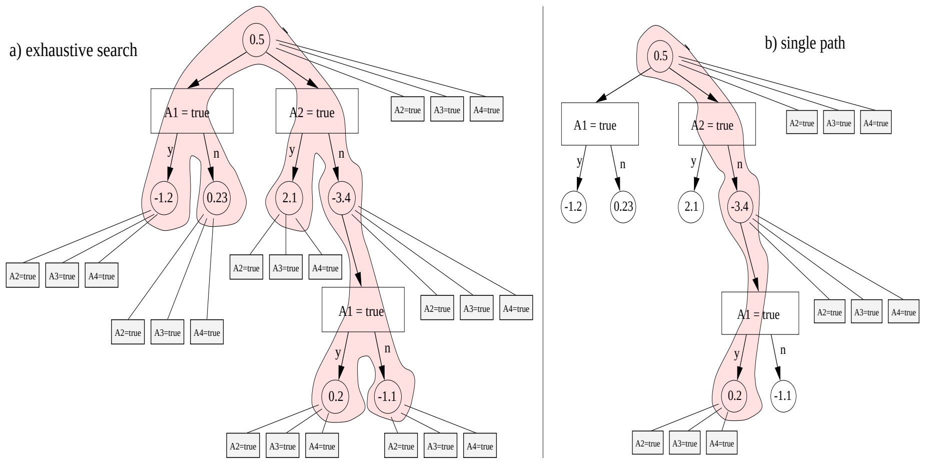

Fig. 2. The exhaustive method (a) has to evaluate all possible additional tests for all predictor nodes. Going down just one path (b) considerably reduces the number of tests to evaluate.

heuristics. The third heuristic was initially added to simply function as a bottom line for comparisons, but turned out to perform pretty well in practise, too.

1. Heaviest path: looking at the formula for Z we see that larger sets of more important examples, i.e. "heavier" sets can lead to a larger reduction, provided we find a test that separates both classes reasonably well. Therefore this heuristic always follows the path of the heaviest[2] subset of examples.
2. Best possible Z_{pure}: reflecting on the previous heuristic we see that it sometimes might lead us astray. The heaviest subset could consist of large a number of examples of just one class, so every conceivable split would still not result in a particularly good Z value. Consequently, this heuristic chooses to follow down the path of the subset with the smallest possible value for Z_{pure}. Clearly, it too cannot provide any guarantees on whether we will be able to find such a split performing as well as theoretically possible.
3. Random walk: this heuristic is a bottom line for comparison and it involves the least computational effort of all. Interestingly, as its choices are purely random, it will explore all paths with equal probability. So every single path has a fair chance of being chosen for evaluation at some boosting iteration or the other.

No matter which method we choose for selecting a single path, we will always be exploring considerably less predictor nodes, which should at least result in considerable savings in terms of time needed for induction. We will try to quantify these savings empirically in the next section.

[2] Heaviest is literally correct here as we sum the weights of the subset of examples at a node to determine how heavy it is.

5 Experiments and Results

Table 1. Datasets used for the experiments

Dataset	Instances	Missing values (%)	Numeric attributes	Nominal
UCI Datasets				
breast-cancer	699	0.2	9	0
cleveland	303	0.2	6	7
credit	690	0.6	6	9
diabetes	768	0.0	8	0
hepatitis	155	5.4	6	13
hypothyroid	3772	5.4	7	22
ionosphere	351	0.0	34	0
kr-vs-kp	3196	0.0	0	36
labor	57	33.6	8	8
mushroom	8124	1.3	0	22
promoters	106	0.0	0	57
sick-euthyroid	3163	6.5	7	18
sonar	208	0.0	60	0
splice	3190	0.0	0	61
vote	435	5.3	0	16
vote1[3]	435	5.5	0	15
KDD Datasets				
coil	5822/4000	0.0	85	0
adult	32561/16281	0.2	6	8
art1	50000/50000	0.0	0	50
art2	50000/50000	0.0	25	25
art3	50000/50000	0.0	50	0

This section compares the performance of the original optimized algorithm of Section 3 with the heuristic variants described in Section 4. The methods were compared for accuracy, runtime and the number of leaves they produced in the resulting tree.

The datasets and their properties are listed in Table 1. The first sixteen are taken from the UCI repository [1]. These datasets were evaluated using a single ten-fold cross validation. The remaining datasets are labelled as "Knowledge Discovery" datasets and come from two sources. The sets called `adult` and `coil` are from the KDD section of the UCI repository while the artificial datasets `art1`, `art2`, and `art3` were generated using a technique described in [5]. Due to their size, these datasets were evaluated using a single train and test split. The table lists the respective train and test set sizes for these cases. One aim of the experiments is to show that the effects of the heuristics scale well with the data.

Figure 3a shows the effect on average relative runtimes of optimizing the original algorithm by merging common branches and employing the Z_{pure} cutoff across all the UCI datasets in Table 1. The figures for the four variations are shown relative to the original algorithm runtime at 10 iterations. The variations are, the original algorithm with no optimization, the original merging common branches only, the original employing the Z_{pure} cutoff only and finally the original using both optimizations. For numbers of boosting iterations up to 50 there is little to be gained by these methods, but beyond 50 significant gains can be made, particularly by merging. The biggest reduction occurs at 200 iterations when both optimizations are used, making an approximate average runtime saving of around one third.

Figure 3b charts the average relative runtimes across the same datasets comparing the original optimized algorithm with the three heuristic methods described in Section 4. The figures for heuristic improvements are relative to "random search" at 10 iterations. The relative differences in performance are only negligible at 10 iterations. Beyond this value the heuristic methods are clearly superior. The random walk method especially is twice as fast as the other two heuristic methods at all iterations, and an order of magnitude faster than the original algorithm at 100 and 200 iterations. In general, the heaviest path and Z_{pure} heuristics have a rather similar runtime behaviour, sometimes they even induced identical trees. The random walk method follows a runtime curve which we suspect is linear. A possible explanation for this surprising average case behaviour is given in the next section.

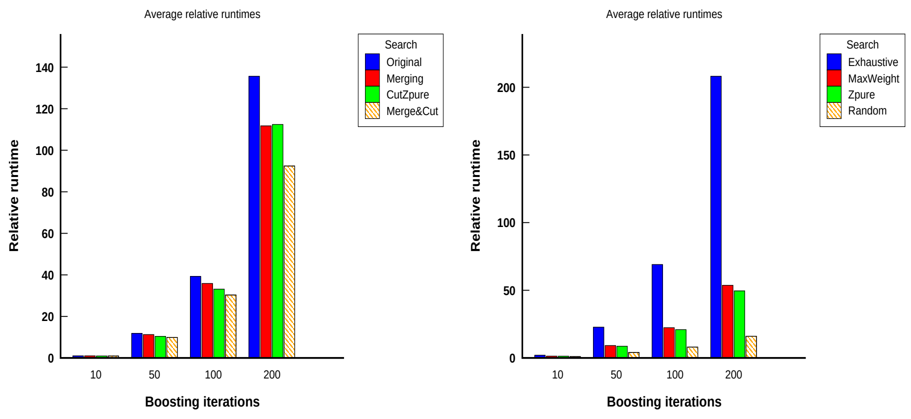

Fig. 3. Average relative runtimes for (a) variations on the original algorithm and (b) the various heuristic search methods.

The runtime performance of the heuristic variants is only relevant if there is no appreciable degradation in predictive accuracy for these methods when compared to the original. Figure 4a shows their performance relative to the original. All error figures are shown relative to the original at 10 iterations. It can be seen that for 10 iterations the heuristic methods fail to produce the same performance as the original, particularly the random walk heuristic which is up to 20% worse. At 50 iterations the gap has closed considerably and beyond 50 the random walk method actually outperforms the original algorithm. The other two heuristics again have the same relative performance which is consistently slightly worse than the original. One explanation for the superior performance of the random walk method is that it may avoid overfitting due to "natural" pruning, but this is only an hypothesis.

The number of leaves (predictor nodes) produced by the various methods give some indication of the shape of the trees being produced by the heuristic methods. Figure 4b shows these leaf figures relative to the original algorithm at the respective number of iterations. As can be seen, the heuristic methods have significant numbers of additional predictor nodes relative to the original method. This is a clear indication that these trees are more shallow, i.e. that they contain more but shorter paths on average. To understand this result we need to visualize the possible shapes of an alternating decision tree. For a fixed total number N of predictor nodes the minimum number of leaves $\frac{N}{2}$ is achieved by a perfectly binary tree. The maximum number $N-1$ is achieved by a flat list of tests; such a totally flat alternating decision tree is actually equivalent to an ensemble of boosted decision stumps. Thus a higher number of leaves indicates a more decision stump-like tree shape.

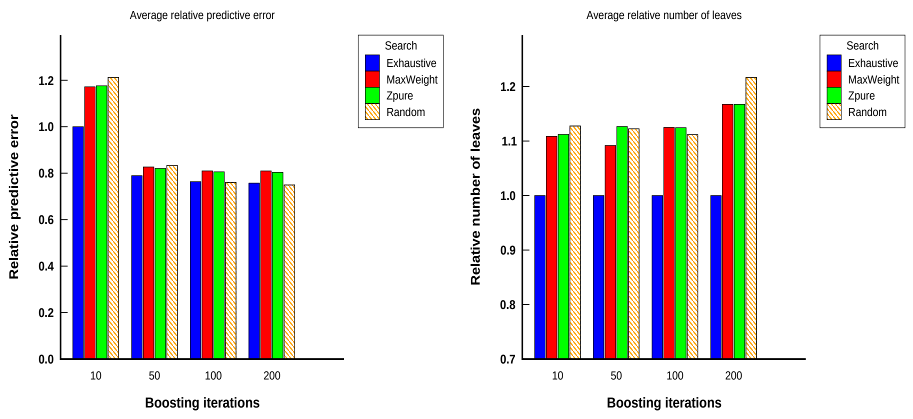

Fig. 4. Average relative accuracies and number of leaves for the various heuristic search methods.

6 Conclusions and Further Work

This paper has presented an improved version of the original alternating decision tree algorithm of Freund and Mason. This improved method is still quadratic in the number of boosting iterations and as such is not particularly useful for knowledge discovery. The use of heuristics to speed up the algorithm was investigated and we have shown that it is possible to achieve results similar, and occasionally better than the original, particularly for large numbers of boosting iterations. In terms of runtime, all heuristic methods were superior. Informal analysis would indicate that all heuristic methods have $O(n \log n)$ worst case complexity, but that they enjoy $O(n)$ average case complexity thanks to the shallowness of the trees they are inducing.

This perceived shallowness also has some impact on comprehensibility. The heuristic methods need to induce larger trees to be competitive with respect to predictive accuracy. Obviously, larger trees are harder to read. But due to the additive nature of alternating decision trees, they can be understood as the sum of all paths. Consequently, we can look at single paths in isolation to understand their respective contribution to the final prediction. Luckily, in a shallow tree most of these paths are rather short, thus they will be relatively easy to comprehend.

In future work we will investigate other approaches on speeding up the original algorithm, which will be based on adaptive caching of some of the statistics that are currently recomputed over and over again. Furthermore, the alternating decision tree algorithm can be extended in a variety of ways. The first and most important is to produce a version of the algorithm capable of handling multiple classes. It would also make sense to apply the trees to regression and cost-sensitive classification problems. Unlike standard decision trees where combining is multiplicative, combining alternating trees is linear which opens up the possibility of being able to bag [2] them to hopefully perform well. Especially in the presence of noise which is problematic for boosting algorithms in general [3], such a bagging approach might alleviate boosting's tendency to overfit the noise.

The improved ADTree induction algorithm as well as the artificial dataset generator described above will both be included into the next version of the WEKA machine learning workbench [9], which is available[4] under the Gnu Public License.

References

1. Blake, C. L., Keogh, E., Merz, C.J.: UCI Repository of Machine Learning DataBases. Irvine, CA: University of California, Department of Information and Computer Science. [http://www.ics.uci.edu/ mlearn/MLRepository.html] (1998).
2. Breiman L.: Bagging Predictors, Machine Learning, 24(2), 1996.

[4] WEKA can be downloaded from `http://www.cs.waikato.ac.nz/~ml`

3. Dietterich T.G.: An Experimental Comparison of Three Methods for Constructing Ensembles of Decision Trees: Bagging, Boosting, and Randomization, Machine Learning, 40(2), 139-158, 2000.
4. Freund, Y., Mason, L.: The alternating decision tree learning algorithm. Proceedings of the Sixteenth International Conference on Machine Learning, Bled, Slovenia, (1999) 124-133.
5. Pfahringer, B., Bensusan, H., Giraud-Carrier, C.: Meta-Learning by Landmarking Various Learning Algorithms. Proceedings of the Seventeenth International Conference on Machine Learning, Stanford University, California, USA (2000) 743-750.
6. Quinlan, J.R.: MiniBoosting Decision Trees. Draft (1999) (available at `http://www.cse.unsw.EDU.AU/~quinlan/miniboost.ps`).
7. Schapire, R.E., Singer, Y.: Improved boosting algorithms using confidence-rated predictions. Machine Learning **37** (3) (1999) 297-336.
8. Witten, I.H., Bell, T.C.: The zero-frequency problem: estimating the probabilities of novel events in adaptive text compression. IEEE Transactions on Information Theory 37(4) (1991) 1085-1094.
9. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann Publishers, San Francisco, California (2000).